Capítulo 5

Herramientas de Desarrollo

Nos acercaremos al uso de diversas herramientas útiles para el programador.

Existen entornos integrados para el desarrollo de software pero no son de uso común. Clásicamente en UNIX se hace uso de diversas herramientas que cumplen con las diversas etapas del desarrollo.

Esta parte del curso es eminentemente práctica. El alumno deberá consultar la ayuda disponible. Como método rápido de consulta, se dispondrá de trípticos con resúmenes útiles de varias herramientas.

5.1 Editor

Vamos a ver tres editores tipo. El alumno deberá escoger aquella opción que más se adecue a sus necesidades.

En primera instancia, se recomienda que use el `pico`.

`pico` ——— *Simple Text Editor*

Es el editor que internamente usa la herramienta para correo electrónico `pine`. Es muy sencillo de usar pero poco potente.

A modo de introducción a su uso realice las siguientes tareas:

(pico) T 5.1

Arranca el editor.

Ctrl G T 5.2

Accede a la ayuda para aprender los pocos controles que tiene.

Ctrl X T 5.3

Abandona el editor.

`vi` ——— *VIsual Editor*

El `vi` es el editor más clásico y lo encontraremos en cualquier máquina UNIX. Es por eso que, a pesar de ser poco amigable, hay mucha gente que lo prefiere a otros más sofisticados. Existe una versión más moderna y potente llamada `vim`.

`vim` T 5.4

Arranca el editor.

T 5.5 Esc

Cambia a "modo mandato".

T 5.6 *:help*

Accede a la ayuda navegable.

T 5.7 *:q*

Abandona la edición actual.

T 5.8 `vimtutor`

Arranca el editor en la realización de un tutorial del mismo.

`emacs` — *GNU project Editor*

El `emacs`, desarrollado por FSF bajo licencia GNU, es mucho más que un simple editor. Es un entorno sobre el que se integran múltiples aplicaciones, correo electrónico, navegador web, news, etc., etc., etc. Está escrito sobre un intérprete de *lisp*.

Existe una versión llamada `Xemacs` más adaptada al entorno gráfico X, y totalmente compatible con el anterior.

T 5.9 `emacs`

Arranca el editor.

T 5.10 Ctrl G

Aborta la acción o mandato a medio especificar.

T 5.11 Ctrl H T

Accede a la realización de un tutorial.

T 5.12 Ctrl X Ctrl C

Abandona la edición actual.

Ejercicios

Para el editor de su elección, o para cada uno de ellos, realice los siguientes ejercicios prácticos:

T 5.13 ☑ *Arranque el editor para crear un nuevo fichero.*

T 5.14 ☑ *Lea el fichero de ejemplo "estilos.c" desde el editor.*

T 5.15 ☑ *Sin modificar su funcionalidad, formatee el texto con el estilo que considere más legible.*

T 5.16 ☑ *Guarde el texto editado con el nombre "mi_estilo.c".*

T 5.17 ☑ *Abandone el editor.*

5.2 Compilador

Compilar un lenguaje de alto nivel, lease C, es traducir el fichero o ficheros fuente en ficheros objeto, que contienen una aproximación al lenguaje máquina, más una lista de los símbolos externos referenciados pero no resueltos.

La etapa de montaje es capaz de enlazar unos ficheros objeto con otros, y a su vez con las bibliotecas necesarias, resolviendo todas las refefrencias a símbolos externos. Esta etapa produce finalmente un fichero ejecutable.

Un fichero ejecutable tiene un formato interno que el sistema operativos es capaz de entender, que describe como situar el proceso en memoria para su ejecución.

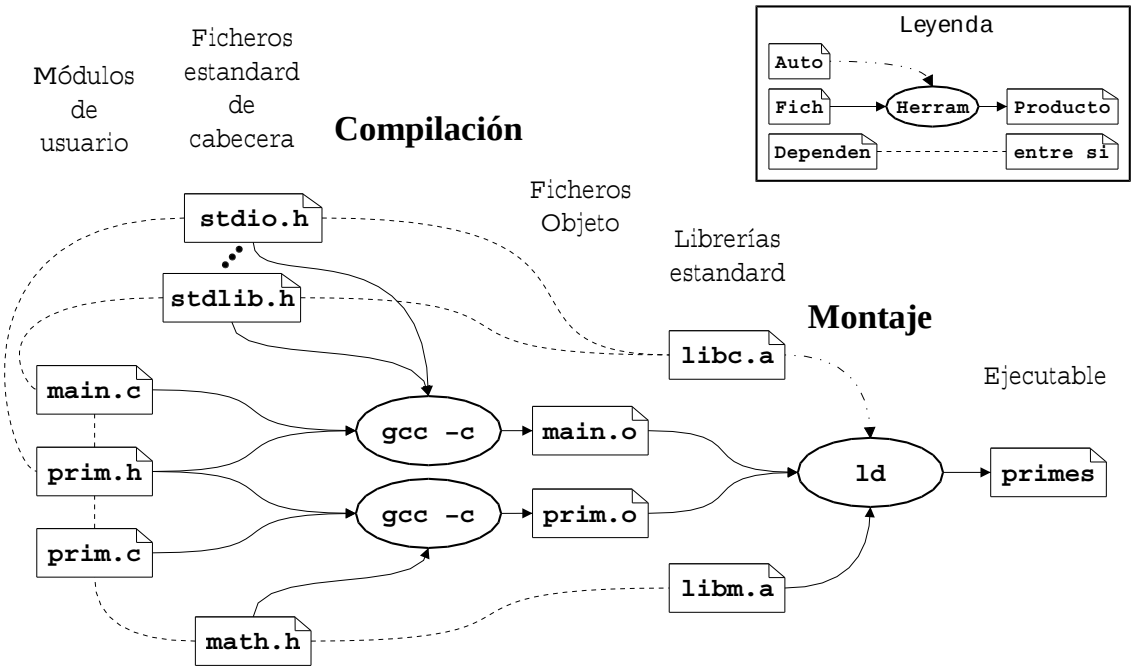

Figura 5.1: Fases de compilación y montaje.

En la figura 5.1 se observa la compilación separada de dos módulos `main.c` y `prim.c` y el montaje de los dos ficheros objeto resultantes para la obtención del fichero ejecutable `primes`.

Extensiones

Ciertos sufijos o extensiones de los nombres de fichero, le indican al compilador los contenidos del mismo, y por lo tanto el tipo de tratamiento que debe realizarse:

`.c` Fuente de C. Debe ser preprocesado, compilado y ensamblado.

`.h` Fichero de cabecera de C. Contiene declaraciones de tipos de datos y prototipos de funciones. Debe ser preprocesado.

`.o` Fichero objeto. Es el resultado de la compilación de un fichero fuente. Debe ser montado.

`.a` Biblioteca de ficheros objeto. Se utiliza para resolver los símbolos de las funciones estandard que hemos usado en nuestro programa y que por lo tanto aún quedan sin resolver en los ficheros objeto.

Errores y *Warnings*

Como ya habrá constatado, los lenguajes de programación, y entre ellos el C, presenta una sintasis bastante estricta. Una de las misiones básicas del compilador es indicarnos los errores que hayamos cometido al codificar, dónde están y en qué consisten.

Los avisos que un compilador es capaz de notificar son de dos tipos:

Errores. Son problemas graves, tanto que evitan que el proceso de compilación pueda concluir. No obstante el compilador seguirá procesando el fichero o ficheros, en un intento de detectar todos los errores que haya.

Es imprescindible corregir estos errores para que el compilador puede terminar de hacer su trabajo.

Warnings. Son problemas leves o alertas sobre posibles problemas en nuestro código.

Aunque el compilador podrá terminar de hacer su trabajo, **es imprescindible corregir los problemas que originan estos warnings**, porque seguramente serán fuente de futuros problemas más graves.

La compilación de un programa correctamente realizado no debería dar ningún tipo de mensaje, y terminar límpiamente.

`gcc` — *GNU C and C++ Compiler*

El nombre que suele recibir el compilador de lenguaje C en UNIX es `cc`. Nosotros usaremos el `gcc` que es un compilador de C y C++, desarrollado por FSF bajo licencia GNU.

Es un compilador muy bueno, rápido y eficiente. Cubre las etapas de compilación y montaje. Parte de uno o más fuentes en C, objetos y/o bibliotecas y finalmente genera un ejecutable autónomo.

Admite multitud de opciones, entre las cuales cabe destacar:

`-c` *file.c* Realiza tan solo el paso de compilación del fichero indicado, pero no el de montaje.

`-o` *name* Solicita que el resultado de la compilación sea un fichero con este nombre.

`-Wall` Activa la detección de todo tipo de posibles errores. El compilador se comvierte en más estricto.

`-g` Añade al fichero objeto o ejecutable información que (como veremos más adelante) permitirá la depuración simbólica del mismo.

`-O` Activa mecanismos de optimización que conseguirán un ejecutable más rápido, a costa de una compilación más larga.

`-l`*library* Solicita el montaje contra la bibliografía especificada.

`-L`*directory* Indica un directorio donde buscar las bibliografías indicadas con la opción `-l`.

Tiene muchisimas más opciones. Consulte el manual si lo precisa.

`minimo.c` T 5.18

Escriba el fichero con el editor de su preferencia y los contenidos indicados en el capítulo anterior 2.3.

`gcc minimo.c` T 5.19

Usamos el compilador para obtener el ejecutable resultante de este fuente.

`ls` T 5.20

Observe que si no le indicamos al compilador otra cosa, el ejecutable resultante tomará el nombre por defecto `a.out`.

`./a.out` T 5.21

Ejecutamos el fichero, indicando que se encuentra en el directorio actual. Esto será necesario si no tenemos el directorio "." en la variable de entorno PATH.

Ahora vamos a trabajar un programa más complejo.

`factorial.c` T 5.22

Escriba el fichero con el editor de su preferencia y los contenidos indicados en el capítulo anterior. El objetivo será mostrar el factorial de los números de 0 al 20 usando un bucle.

`gcc -g -c factorial.c` T 5.23

Con este paso compilaremos, para obtener el fichero objeto `factorial.o`. *Este fichero contendrá información simbólica.*

`gcc factorial.o -o factorial` T 5.24

Cuando compile correctamente, montaremos para obtener el ejecutable.

`./factorial` T 5.25

Ejecute el mandato y compruebe su correcto funcionamiento.

5.3 Depurador

La compilación sin mácula de un programa no implica que sea correcto. Es muy posible que contenga errores que sólo se harán visibles en el momento de su ejecución. Para corregir estos errores necesitamos una herramienta que nos permita supevisar la ejecución de nuestro programa y acorralar el error hasta localizarlo en el código.

Existen depuradores de bajo y de alto nivel. Los de bajo nivel nos permiten inspeccionar el código máquina o ensamblador, y su ejecución paso a paso. Los alto nivel, tambien llamados depuradores simbólicos, permiten visualizar y controlar la ejecución línea a línea del código fuente que hemos escrito, así como inspeccionar el valor de las variables de nuestro programa..

Para que un programa pueda ser depurado correctamente, su ejecutable deberá contener información que lo asocie al código fuente del que derivó. Para ello deberá haber sido compilado con la opción `-g`.

Algunas de las funciones del depurador son:

- Establecer puntos de parada en la ejecución del programa (*breakpoints*).
- Examinar el valor de variables
- Ejecutar el programa línea a línea

Nosotros usaremos el `gdb` desarrollado por FSF bajo licencia GNU.
Tambien veremos en `ddd` que es un *frontend* gráfico para el `gdb`.

Fichero core

Trabajando en UNIX nos sucederá a menudo, que al ejecutar un programa este termine bruscamente y apareciendo algún mensaje de error como por ejemplo:
`segmentation fault:  core dumped`.

Este mensaje nos informa de que nuestro programa en ejecución, esto es, el proceso, ha realizado un acceso ilegal a memoria y el sistema operativo lo ha matado. Así mismo nos indica que se ha generado un fichero de nombre `core`. El fichero `core` es un volcado de la memoria del proceso en el preciso instante en que cometío el error, esto es una instantánea. Este fichero nos servirá para poder estudiar con todo detalle la situación que dió lugar al error.

`gdb prog [core]` ——————————————— *GNU debugger*

Invocaremos al `gdb` indicádole el programa ejecutable que queremos depurar y opcionalmente indicaremos en fichero `core` que generó su anómala ejecución.

Una vez arrancado el depurador, proporciona una serie de mandatos propios para controlar la depuración:

`help` Da acceso a un menú de ayuda.

`run` Arranca la ejecución del programa.

`break` Establece un *breakpoint* (un número de línea o el nombre de una función).

`list` Imprime las líneas de código especificadas.

`print` Imprime el valor de una variable.

`continue` Continúa la ejecución del programa después de un *breakpoint*.

`next` Ejecuta la siguiente línea. Si se trata de una llamada a función, la ejecuta completa.

`step` Ejecuta la siguiente línea. Si se trata de una llamada a función, ejecuta sólo la llamada y se para al principio de la misma.

`quit` Termina la ejecución del depurador

Para más información sobre el depurador consulte su ayuda interactiva, el manual o el tríptico.

Más adelante tendremos oportunidad de hacer un uso extenso del depurador.

`ddd prog [core]` ——— *Data Display Debugger*

Es un *frontend* gráfico para el `gdb`, esto es, nos ofrece un interfaz gráfico bastante intuitivo para realizar la depuración, pero en definitiva estaremos usando el `gdb`. Como característica añadida, nos ofrece facilidades para la visualización gráfica de las estructuras de datos de nuestros programas, lo cuál será muy útil. Para usar el `ddd` es presiso tener arrancadas las X-Windows.

5.4 Bibliotecas

Una regla básica para realizar software correcto es utilizar código que ya esté probado, y no estar reinventando sistemáticamente la rueda. Si existe una función de biblioteca que resuelve un problema deberemos usarla y no reescribir tal funcionalidad.

Existen muchas muchas bibliotecas con facilidades para realizar todo tipo de cosas: aritmética de múltiple precisión, gráficos, sonidos, etc., etc.

Para poder hacer uso correcto de esas funciones de biblioteca debemos consultar el manual que nos indicará la necesidad de incluir en nuestro fichero fuente cierto fichero de cabecera (ej. `#include <string.h>`), que contiene las declaraciones de tipos de datos y los prototipos de las funciones de biblioteca.

Se recomienda que consulte el triptico de ANSI C.

Destacaremos aquí dos bibliotecas fundamentales:

`libc.a` Es la biblioteca estandard de C. Contiene funciones para: manejo de tiras de caracteres, entrada y salida estandard, etc.

Se corresponde con las funciones definidas en los ficheros de cabecera:

```
<assert.h>  <ctype.h>  <errno.h>   <float.h>   <limits.h>
<locale.h>  <math.h>   <setjmp.h>  <signal.h>  <stdarg.h>
<stddef.h>  <stdio.h>  <stdlib.h>  <string.h>  <time.h>
```

El montaje contra esta biblioteca se realiza de forma automática. No es preciso indicarlo, pero lo haríamos invocando al compilador con la opción `-lc`.

`libm.a` Es la biblioteca que contiene funciones de cálculo matemático y trigonométrico: `sqrt`, `pow`, `hypot`, `cos`, `atan`, etc. . Para hacer uso de las funciones de esta biblioteca habría que incluir en nuestro fichero `<math.h>`.

El montaje contra esta biblioteca se realiza invocando al compilador con la opción `-lm`.

`ar -opts [member] arch files...` ——— *Manage Archive Files*

Cuando desee realizar medianos o grandes proyectos, podrá realizar sus propias bibliotecas utilizando esta herramienta. Es una utilidad para la creación y mantenimiento de bibliotecas de ficheros.

Para usar una biblioteca, se especifica en la línea de compilación la biblioteca (por convención `libnombre.a`) en vez de los objetos.

Algunas opciones frecuentemente usadas son:

`-d` Elimina de la biblioteca los ficheros especificados

`-r` Añade (o reemplaza si existe) a la biblioteca los ficheros especificados. Si no existe la biblioteca se crea

`-ru` Igual que -r pero sólo reemplaza si el fichero es más nuevo

`-t` Muestra una lista de los ficheros contenidos en la biblioteca

`-v` *Verbose*

`-x` Extrae de la biblioteca los ficheros especificados

A continuación se muestran algunos ejemplos del uso de este mandato.

Para obtener la lista de objetos contenidos en la biblioteca estándar de C, se ejecutaría:

```
ar -tv /usr/lib/libc.a
```

El siguiente mandato crea una biblioteca con objetos que manejan distintas estructuras de datos:

```
ar -rv $HOME/lib/libest.a pila.o lista.o
```

```
ar -rv $HOME/lib/libest.a arbol.o hash.o
```

Una vez creada la biblioteca, habría dos formas de compilar un programa que use esta biblioteca y además la matemática:

```
cc -o pr pr.c -lm $HOME/lib/libest.a
```

```
cc -o pr pr.c -lm -L$HOME/lib -lest
```

5.5 Constructor

Si el programa o aplicación que estamos desarrollando está convenientemente descompuesto en múltiples módulos, el proceso global de compilación puede ser automatizado haciendo uso de la herramienta `make`.

`make` _____ *Aplication Mantainer*

Esta utilidad facilita el proceso de generación y actualización de un programa. Determina automáticamente qué partes de un programa deben recompilarse ante una actualización de algunos módulos y las recompila. Para realizar este proceso, `make` debe conocer las dependencias entre los ficheros: un fichero debe actualizarse si alguno de los que depende es más nuevo.

La herramiente `make` consulta un fichero (`Makefile`) que contiene las reglas que especifican las dependencias de cada fichero objetivo y los mandatos para actualizarlo. A continuación, se muestra un ejemplo:

Makefile

```
# Esto es un comentario
CC=gcc                  # Esto son macros
CFLAGS=-g
OBJS2=test.o prim.o

all: primes test        # Esta es la primera regla

primes: main.o prim.o   # Esta es otra regla
       gcc -g -o primes main.o prim.o -lm
       # Este es el mandato asociado

test: $(OBJS2)          # Aquí usamos las macros
       ${CC} ${CFLAGS} -o $@ ${OBJS2}

main.o prim.o test.o : prim.h # Esta es una dependencia.

clean:  # Esta no depende de nada, es obligatoria.
       rm -f main.o ${OBJS2}
```

Observe que las líneas que contienen los mandatos deben estar convenientemente tabuladas haciendo uso del tabulador, no de espacios. De no hacerlo así, la herramienta nos indicará que el formato del fichero es erróneo.

Ejecutaremos `make` para que se dispare la primera regla, o `make clean` explicitando la regla que queremos disparar. Entonces lo que sucederá es:

1. Se localiza la regla correspondiente al objetivo indicado.

2. Se tratan sus dependencias como objetivos y se disparan recursivamente.

3. Si el fichero objetivo es menos actual que alguno de los ficheros de los que depende, se realizan los mandatos asociados para actualizar el fichero objetivo.

Existen macros especiales. Por ejemplo, $@ corresponde con el nombre del objetivo actual. Asimismo, se pueden especificar reglas basadas en la extensión de un fichero. Algunas de ellas están predefinidas (p.ej. la que genera el .o a partir del .c).

Ejercicio 5.1.

5.6 Otras herramientas

Existen variedad de otras herramientas que podrán serle de utilidad cuando desarrolle código en un sistema UNIX Citaremos tres:

`gprof`
: Es una herramienta que nos permite realizar un perfil de la ejecución de nuestros programas, indicando dónde se pierde el tiempo, de menera que tendremos criterio para optimizar si es conveniente estas zonas de nuestro código.

`gcov`

Es semejante a la anterior, pero su objetivo es distinto. En la fase de verificación del programa, permite cubrir el código, es decir, asegurar que todas las líneas de código de nuestro programa han sido probadas con suficiencia.

`indent`

Permite indentar el ficheros fuente en C. Es muy parametrizable, para que el resultado final se corresponda con el estilo del propio autor.